Testing with Python

Valentin Haenel
Strongly based on material by Pietro Berkes

Freelance Consultant and Software Developer
http://haenel.co

Feb 2013

Outline

Testing with Python

- `unittest`:
  - Has been part of the Python standard library since v. 2.1
  - Interface a bit awkward (camelCase methods...), very basic functionality until...
  - Major improvement with 2.7, now at the level of other modern testing tools
  - Backward compatible, unittest2 back-port for earlier versions of Python
- alternatives:
  - `nosetests`
  - `py.test`
  - `doctest`

Test suites in Python: unittest

- Each test case is a subclass of `unittest.TestCase`
- Each test unit is a method of the class, whose name starts with `test`
- Each test unit checks **one** aspect of your code, and raises an exception if it does not work as expected

Anatomy of a TestCase

```python
import unittest

class FirstTestCase(unittest.TestCase):

    def test_truisms(self):
        """All methods beginning with 'test' are executed"""
        self.assertTrue(True)
        self.assertFalse(False)

    def test_equality(self):
        """Docstrings are printed during executions
        of the tests in some test runners"""
        self.assertEqual(1, 1)

if __name__ == '__main__':
    unittest.main()
```

Multiple TestCases

```python
import unittest

class FirstTestCase(unittest.TestCase):

    def test_truisms(self):
        self.assertTrue(True)
        self.assertFalse(False)

class SecondTestCase(unittest.TestCase):

    def test_approximation(self):
        self.assertAlmostEqual(1.1, 1.15, 1)

if __name__ == '__main__':
    # execute all TestCases in the module
    unittest.main()
```

TestCase.assertSomething

- `TestCase` defines utility methods to check that some conditions are met, and raise an exception otherwise
- Check that statement is true/false:
  - `assertTrue('Hi'.islower())`

    ⇒ fail
  - `assertFalse('Hi'.islower())` ⇒ pass
- Check that two objects are equal:
  - `assertEqual(2+1, 3)` ⇒ pass
  - `assertEqual([2]+[1], [2, 1])` ⇒ pass
  - `assertNotEqual([2]+[1], [2, 1])` ⇒ fail
- `assertEqual` can be used to compare numbers, lists, tuples, dicts, sets, frozensets, and unicode objects

- Check that two numbers are equal up to a given precision:
  - `assertAlmostEqual(x, y, places=7)`
- `places` is the number of decimal places to use:
  - `assertAlmostEqual(1.121, 1.12, 2)` ⇒ pass
  - `assertAlmostEqual(1.121, 1.12, 3)` ⇒ fail

Formula for almost-equality is:

`round(x - y, places) == 0.`

And so...

`assertAlmostEqual(1.126, 1.12, 2)` ⇒ fail

TestCase.assertSomething

- One can also specify a maximum difference:
  - `assertAlmostEqual(x, y, delta=0.)`
- E.g.:
  - `assertAlmostEqual(1.125, 1.12, 0.06)` $\Rightarrow$ pass
  - `assertAlmostEqual(1.125, 1.12, 0.04)` $\Rightarrow$ fail
- Can be used to compare any object that supports subtraction and comparison:

```
import datetime

delta = datetime.timedelta(seconds=10)
second_timestamp = datetime.datetime.now()
self.assertAlmostEqual(first_timestamp,
        second_timestamp, delta=delta)
```

TestCase.assertSomething

- Check that an exception is raised:

`assertRaises(exception_class, function, args, kwargs)`

- executes:

`function(args, kwargs)`

- and passes if an exception of the appropriate class is raised
- For example:
  - `assertRaises(IOError, file, 'inexistent', 'r')` ⇒ pass
- Use the most specific exception class, or the test may pass because of collateral damage:
  - `tc.assertRaises(IOError, file, 1, 'r')` ⇒ fail
  - `tc.assertRaises(Exception, file, 1, 'r')` ⇒ pass

TestCase.assertSomething

- The most convenient way to use assertRaises is as a context manager (`with` statement):

```
with self.assertRaises(SomeException):
        do_something()
```

- For example:

```
with self.assertRaises(ValueError):
        int('XYZ')
```

- passes, because

```
int('XYZ')
ValueError: invalid literal for int() with base 10: 'XYZ'
```

`TestCase.assertSomething`

- Many more `assert` methods: (complete list)
- `assertGreater(a, b) / assertLess(a, b)`
- `assertRegexpMatches(text, regexp)`
  - verifies that regexp search matches text
- `assertIn(value, sequence)`
  - assert membership in a container
- `assertIsNone(value)`
  - verifies that value is None
- `assertIsInstance(obj, cls)`
  - verifies that an object is an instance of a class
- `assertItemsEqual(actual, expected)`
  - verifies equality of members, ignores order
- `assertDictContainsSubset(subset, full)`
  - tests whether the entries in dictionary full are a superset of those in subset

TestCase.assertSomething

- Most of the assert methods accept an optional `msg` argument that overwrites the default one:

```
assertTrue('Hi'.islower(), 'One of the letters is not lowercase')
```

- Most of the assert methods have a negated equivalent, e.g.:
  - `assertIsNone`
  - `assertIsNotNone`

Doctests

- `doctest` is a module that recognizes Python code in documentation and tests it
  - Can be in docstrings, rst or plain text documents
  - Serves as an example to the reader **and** as test code

Syntax

```
>>> CODE
EXPECTED\_RESULT
```

Doctest Example

```
def my_max(iterable):
    """ find the maximum in an iterable

    >>> my_max([1])
    1
    >>> my_max([1, 2])
    2
    >>> my_max([-1, -2])
    -1
    """
    best = iterable[0]
    for i in iterable[1:]:
        if i > best:
            best = i
    return best
```

Testing with Numpy arrays

- When testing numerical algorithms, Numpy arrays have to be compared elementwise:

```
import unittest, numpy

class NumpyTestCase(unittest.TestCase):

    def test_equality(self):
        a = numpy.array([1, 2])
        b = numpy.array([1, 2])
        self.assertEqual(a, b)
```

because...

Testing with Numpy arrays

```
$ nosetest test_numpy.py
E
======================================================================
ERROR: test_equality (test_numpy.NumpyTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/esc/git-working/python-cuso/testing/code/test_numpy.py",
  line 9, in test_equality
    self.assertEqual(a, b)
  File "/usr/lib/python2.6/unittest.py", line 348, in failUnlessEqual
    if not first == second:
ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()

----------------------------------------------------------------------
Ran 1 test in 0.032s

FAILED (errors=1)
```

Testing with Numpy arrays

- `numpy.testing` defines appropriate function:
  - `numpy.testing.assert_array_equal(x, y)`
  - `numpy.testing.assert_array_almost_equal(x, y, decimal=6)`
- If you need to check more complex conditions:
  - `numpy.all(x)`
    - returns `True` if all elements of `x` are true
  - `numpy.any(x)`
    - returns `True` is any of the elements of `x` is true
  - `numpy.allclose(x, y, rtol=1e-05, atol=1e-08)`
    - returns `True` if two arrays are element-wise equal within a tolerance; `rtol` is relative difference, `atol` is absolute difference
  - combine with `logical_and`, `logical_or`, `logical_not`:

```
# test that all elements of x are between 0 and 1
assertTrue(all(logical_and(x > 0.0, x < 1.0)))
```

Outline

How to run tests with unittest

- Option 1: `unittest.main()` will execute all tests in all `TestCase` classes

```
if __name__ == '__main__':
        unittest.main()
```

- Option 2: Execute all tests in one file

```
$ python -m unittest [-v] <test_module>
```

- Option 3: Discover all tests in all subdirectories

```
$ python -m unittest discover
```

Running doctests

- Option 1: use `doctest.testmod()`

```python
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

- Option 2: Execute all tests in one file

```
$ python -m doctest <module>
```

Using the `nosetests` runner

- Performs automatic discovery
- Fully compatible with the `unittest` package
- Uses various heuristics to look for tests

```
$ nosetests
................................
----------------------------------------------------------------------
Ran 32 tests in 4.876s

OK
```

- In my experience: it does the right thing (TM)

Options for the `nosetests` runner

- Has a myriad of options, e. g.
  - `-v, --verbose`: print the name of each test as it is run
  - `--pdb`: drop directly into the debugger on failure
  - `--processes=NUM`: split testing into multiple processes (automatic parallelization)
- And a myriad of plugins, e. g.
  - `--with-coverage`: run the tests with `coverage.py` to measure test coverage
  - `--with-doctest`: run any doctests too!

Running single tests with the nosetests runner

- ... And a very useful way to run single test classes and/or functions:

```
$ nosetests <FILE>:<FUNCTION>
$ nosetests <FILE>:<CLASS>.<METHOD>
```

```
$ nosetests test_file.py:test_function
.
----------------------------------------------------------------------
Ran 1 test in 0.208s

OK
```

Outline

Basics of testing

- What to test, and how?
- At the beginning, testing feels weird:
  1. It's obvious that this code works (not TDD...)
  2. The tests are longer than the code
  3. The test code is a duplicate of the real code
- What does a good test looks like?
- What should I test?
- Anything specific to scientific code?

Basic structure of test

- A good test is divided in three parts:
  - **Given:** Put your system in the right state for testing
    - Create objects, initialize parameters, define constants...
    - Define the expected result of the test
  - **When:** The key actions of the test
    - Typically one or two lines of code
  - **Then:** Compare outcomes of the key actions with the expected ones
    - Set of *assertions* regarding the new state of your system

Test simple but general cases

- Start with simple, general case
- Take a realistic scenario for your code, try to reduce it to a simple example
- Tests for `lower` method of strings

```python
import unittest

class LowerTestCase(unittest.TestCase):
    def test_lower(self):
        # Given
        string = 'HeLlO wOrld'
        expected = 'hello world'
        # When
        output = string.lower()
        # Then
        self.assertEqual(output, expected)
```

Test special cases and boundary conditions

- Code often breaks in corner cases: empty lists, None, NaN, 0.0, lists with repeated elements, non-existing file, ...
- This often involves making design decision: respond to corner case with special behavior, or raise meaningful exception?

```
def test_empty_string(self):
    # Given
    string = ''
    expected = ''
    # When
    output = string.lower()
    # Then
    self.assertEqual(output, expected)
```

- Other good corner cases for string.lower():
  - do-nothing case: `string = 'hi'`
  - symbols: `string = '123 (!'`

Common testing pattern

- Often these cases are collected in a single test:

```python
import unittest

class LowerTestCase(unittest.TestCase):
    def test_lower(self):
        # Given
        # Each test case is a tuple of (input, expected_result)
        test_cases = [('HeLlO wOrld', 'hello world'),
                      ('hi', 'hi'),
                      ('123 ([?', '123 ([?'),
                      ('', '')]
        for string, expected in test_cases:
            # When
            output = string.lower()
            # Then
            self.assertEqual(output, expected)
```

Fixtures

- Tests require an initial state or test context in which they are executed (the “Given” part), which needs to be initialized and possibly cleaned up.
- If multiple tests require the same context, this fixed context is known as a *fixture*.
- Examples of fixtures:
  - Creation of a data set at runtime
  - Loading data from a file or database
  - Creation of mock objects to simulate the interaction with complex objects

setUp and tearDown

```python
import unittest

class FirstTestCase(unittest.TestCase):
    def setUp(self):
        """setUp is called before every test"""
        pass
    def tearDown(self):
        """tearDown is called at the end of every test"""
        pass
    @classmethod
    def setUpClass(cls):
        """Called once before all tests in this class."""
        pass
    @classmethod
    def tearDownClass(cls):
        """Called once after all tests in this class."""
        pass
    # ... all tests here ...
```

Numerical fuzzing

- Use deterministic test cases when possible
- In most numerical algorithm, this will cover only over- simplified situations; in some, it is impossible
- Fuzz testing, or **fuzzing**: generate random input
    - Outside scientific programming it is mostly used to stress-test error handling, memory leaks, safety
    - For numerical algorithms, it is used to make sure one covers general, realistic cases
    - The input may be random, but you still need to know what to expect
    - Make failures reproducible by saving or printing the random seed

Numerical fuzzing – example

```
import unittest, numpy

class VarianceTestCase(unittest.TestCase):

    def setUp(self):
        self.seed = int(numpy.random.randint(2**31-1))
        numpy.random.seed(self.seed)
        print 'Random seed for the tests:', self.seed

    def test_var(self):
        N, D = 100000, 5
        # goal variances: [0.1 , 0.45, 0.8 ,    1.15,  1.5]
        desired = numpy.linspace(0.1, 1.5, D)
        # test multiple times with random data
        for _ in range(20):
            # generate random, D-dimensional data
            x = numpy.random.randn(N, D) * numpy.sqrt(desired)
            variance = numpy.var(x, axis=0)
            numpy.testing.assert_array_almost_equal(
                    variance, desired, 1)
```

Other common cases

- Test general routines with specific ones
- Example: test `polyomial_expansion(data, degree)` with `quadratic_expansion(data)`
- Test optimized routines with brute-force approaches

Example (test `z = outer(x, y)` with:)

```
M, N = x.shape[0], y.shape[0]
z = numpy.zeros((M, N))
for i in range(M):
    for j in range(N):
        z[i, j] = x[i] * y[j]
```

Example: eigenvector decomposition

- Consider the function `values, vectors = eigen(matrix)`
- Test with simple but general cases:
  - use full matrices for which you know the exact solution (from a table or computed by hand)
- Test general routine with specific ones:
  - use the analytical solution for 2x2 matrices
- Numerical fuzzing:
  - generate random eigenvalues, random eigenvector; construct the matrix; then check that the function returns the correct values
- Test with corner cases:
  - test with diagonal matrix: is the algorithm stable?
  - test with a singular matrix: is the algorithm robust? Does it raise appropriate error when it fails?

Summary

- Testing is an essential part of modern software development
- In Python all batteries are included and testing is easy
- In fact, because it is a dynamic language testing is the only way to ensure correctness as your program evolves
- Code that is easy to test is usually easy to modify
- If you are already testing your code, give Test Driven Development a try

Conclusion

- Open source tools used to make this presentation:
  - Wiki2beamer
  - LaTeXbeamer
  - Dia
  - Pygments
  - Minted
  - Solarized theme for pygments